ISIS-II
PL/M-80 COMPILER
OPERATOR'S MANUAL

Document Number 9800300C

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

ISIS-II
PL/M-80 COMPILER
OPERATOR'S MANUAL

Document Number 9800300C

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

PREFACE

This manual describes the operation of the PL/M-80 Compiler, version 3.1. The compiler accepts PL/M-80 source as input and produces relocatable 8080 object code as output. The compiler runs under the ISIS-II operating system which supports relocation and linkage of object code programs. The manual is one of a series of documents describing this system and its operation.

This manual assumes that the reader is conversant with PL/M-80, is familiar with the ISIS-II operating system, and knows how to operate the Intellec Microcomputer Development System hardware. The reader is referred to the following Intel publications to gain such familiarity.

- *PL/M-80 Programming Manual* 98-268
- *ISIS-II System Users Guide* 98-306
- *Intellec Microcomputer Development System Operator's Manual* 98-129

The compiler requires the following software and hardware environments for proper execution:

Software:

- ISIS-II Operating System
- ISIS-II Relocation Software
- IXREF Program (if intermodule cross-reference listing is desired)

Hardware:

- 8080 Intellec Microcomputer Development System
- 64K bytes of RAM memory
- Dual flexible diskette drives and controller
- Console device (TTY or CRT)

CONTENTS

ILLUSTRATIONS

CHAPTER 1
HOW TO USE THE PL/M-80 COMPILER

This chapter presents all of the information necessary to begin using the PL/M-80 Compiler. It is not necessary to be familiar with all the features described in the rest of this manual in order to make effective use of the compiler. If you are a beginning user you are particularly encouraged to start using the compiler and to gain experience with PL/M-80 before concerning yourself with special features. The example included in this chapter can be entered exactly as shown to get a feel for the procedures involved in using the compiler.

The compiler is supplied on a diskette which does not contain an operating system or relocation software. It may be desirable to copy the compiler to another diskette (such as a system diskette). Section 2.2.4 lists the files that contain the code of the compiler.

The following example illustrates the normal sequence of operations used to compile a PL/M-80 program from system bootstrap to eventual program execution. The steps involved are as follows:

1. Power up the Intellec hardware.
2. Insert a system diskette into Drive 0. In this example, the system diskette contains the compiler.
3. Insert a nonsystem diskette into Drive 1.
4. Bootstrap the ISIS-II Operating System.
5. Enter the source program to a file on Drive 1 using the EDIT program.
6. Compile the program with the PL/M-80 Compiler.
7. LINK and LOCATE the resulting object code program.
8. Execute the program.

Refer to the *ISIS-II System Users Guide* for detailed instructions for all of these steps with the exception of compiling your program. This manual describes program compilation.

In the interactive sequence shown in Figure 1-1, underlined text is output by the system, all other text is typed by the user. Comments appearing to the right of semicolons are for clarification, not material entered by the user. This example shows how to create, compile, load and execute a complete program, which uses an ISIS-II system library routine to write a message to the console.

In the normal usage of the PL/M-80 Compiler the compilation listing is written by default to a diskette file on the same diskette as the source file. This file has the same name as the source file, but has the extension LST. Thus, in the example above, the listing is found in :F1:MYPROG.LST. Similarly, the object code file is on the same diskette and has the same file name, but has the extension OBJ. In the example :F1:MYPROG.OBJ contains the object code produced by compiling :F1:MYPROG.SRC.

A detailed explanation of all of the steps used in the example, with the exception of the command that invokes the PL/M-80 Compiler, may be found in the *ISIS-II System Users Guide*. See also Section 5.4 of this manual for an explanation of why the PLM80.LIB file must be linked in.

The normal method of invoking the compiler, when no special actions are needed, is simply to give its name (PLM80) and the name of your source file. The source file must be on a diskette and must contain a PL/M-80 source module. This command has the form

```
PLM80 source-file
```

if the compiler is in Drive 0.

In this chapter everything necessary to use the compiler in its normal mode of operation has been shown, and you need not read the remainder of this manual unless you need additional features. The remaining chapters of the manual provide a detailed description of all available compiler features.

```
ISIS-II, V2.2                                  ;the system identifies itself

-EDIT :F1:MYPROG.SRC                           ;the editor is invoked...
                                               ;...to create the source program
ISIS-II TEXT EDITOR, V1.6
NEW FILE
*I
MYPROGRAM:  DO;
  WRITE: PROCEDURE(AFT,BUFFER,LENGTH,STATUS) EXTERNAL;
           DECLARE(AFT,BUFFER,LENGTH,STATUS) ADDRESS;
         END WRITE;
  EXIT:  PROCEDURE EXTERNAL;
         END EXIT;
  DECLARE STATUS ADDRESS,
          CRLF LITERALLY '0AH,0DH';    /*CARRIAGE RETURN, LINE FEED*/

  CALL WRITE(0,.('HELLO, THIS IS A PL/M-80 PROGRAM', CRLF),34,.STATUS);
  CALL EXIT;
END MYPROGRAM;
$E$$                                           ; $ denotes ESCAPE

-PLM80 :F1:MYPROG.SRC                          ;the compiler is invoked

ISIS-II PL/M-80 COMPILER, V3.1
PL/M-80 COMPILATION COMPLETE.      0 PROGRAM ERROR(S)

-LINK :F1:MYPROG.OBJ,SYSTEM.LIB,PLM80.LIB TO MYPROG.LNK

-LOCATE MYPROG.LNK                             ;the linked program is relocated

-MYPROG                                        ;the program is executed
HELLO, THIS IS A PL/M-80 PROGRAM
-
```

Figure 1-1. Interactive Compilation Sequence

CHAPTER 2
COMPILER INVOCATION AND FILE USAGE

Throughout this manual, the following conventions are used in describing the commands and controls associated with the compiler:

- Upper-case letters (and numerals) represent text that must be entered as shown in the description (however, you may enter these items in lower-case).
- Lower-case letters are used to represent variable parts of the command or control.
- Square brackets [ ] are used to enclose parts of the command or control that may be omitted (the brackets themselves are not part of the command or control).

The following discussions assume that the ISIS-II system has been bootstrapped. A diskette containing the PL/M-80 Compiler must be mounted in one of the diskette drivers before the compiler is used. (If it is not in Drive 0, a system diskette must be mounted in Drive 0.)

2.1 COMPILER INVOCATION

The PL/M-80 Compiler is invoked from the ISIS-II console using the standard command format described in the *ISIS-II System Users Guide*.

The invocation command has the general form:

[:device:]PLM80 source-file [controls]

where

- *device* identifies which drive contains the compiler diskette. This may be omitted if the compiler diskette is in Drive 0.
- *source-file* is the name of the file containing the PLM/-80 source module.
- *controls* is an optional sequence of compiler controls. The use of these controls is described in Chapter 3.

Figure 2-1 shows two examples of compiler invocation. In the first example, the compiler is directed to compile the source module on :F1:PROG1.SRC. This file resides on the diskette in Drive 1 and has the name PROG1.SRC. In the second example, the compiler diskette is in Drive 1. The compiler is directed to compile the source module on :F1:MYPROG.SRC, directing all printed output to :LP:, and placing 'TEST PROGRAM 4' in the header on each page of the listing.

2.2 FILE USAGE

2.2.1 INPUT FILES

The compiler reads the PL/M-80 source from the source-file specified on the command line (see previous section) and also from any files specified with INCLUDE controls (see Section 3.7). These files must be standard ISIS-II diskette files. The source input should contain a PL/M-80 source module.

2.2.2 OUTPUT FILES

Two output files are produced during each compilation unless specified controls are used to suppress them. These are the listing and object code files. Each of these may be explicitly directed to some standard ISIS-II path (device or file) by using the PRINT and OBJECT controls respectively. If the user does not control these outputs explicitly, the compiler writes them to disk files on the diskette containing the input file. These files have the same file name as the input file, but have the extensions LST for the listing and OBJ for the object code. For example, if the compiler is invoked by

```
PLM80 :F1:MYPROG.SRC
```

the listing and all other printed output is written to :F1:MYPROG.LST and the object code to :F1:MYPROG.OBJ. If these files already exist they are overwritten. If they do not exist the compiler creates them.

The object code file may be used as input to the ISIS-II relocation and linkage programs, LINK, LOCATE, and LIB.

```
-PLM80 :F1:PROG1.SRC
-:F1:PLM80 :F1:MYPROG.SRC   PRINT(:LP:)   TITLE('TEST PROGRAM 4')
```

Figure 2-1. Two Examples of Compiler Invocation

2.2.3 COMPILER WORK FILES

The compiler uses five work files during its operation which are deleted at the completion of compilation. The user should not use files with these names, as their contents will be destroyed:

```
:F1:PLMTX1.TMP
:F1:PLMTX2 TMP
:F1:PLMAT.TMP
:F1:PLMXRF.TMP
:F1:PLMNMS.TMP
```

NOTE

If the WORKFILES control (see Section 3.6) is not used, all of these files will be on :F1: as shown here. If the WORKFILES control is used, these files may be on other drives.

2.2.4 COMPILER CODE FILES

The compiler's object code resides in six diskette files. These files must be present for proper execution of the compiler:

```
PLM80
PLM80.OV0
PLM80.OV1
PLM80.OV2
PLM80.OV3
PLM80.OV4
```

The diskette containing these files may be mounted in any diskette drive—not necessarily Drive 0.

CHAPTER 3
COMPILER CONTROL LANGUAGE

3.1 INTRODUCTION TO COMPILER CONTROLS

The exact operation of the compiler may be controlled by a number of *controls* which specify options such as the type of listing to be produced and the destination of the object file. Controls may be specified as part of the ISIS-II command invoking the compiler, or as *control lines* appearing as part of the source input file.

A *control line* is a source line containing a dollar sign ($) in the left margin. Normally, the left margin is set at column one, but this may be changed with the LEFTMARGIN control. Control lines are introduced into the source to allow selective control over sections of the program. For example, it may be desirable to suppress the listing for certain sections of a program, or to cause page ejects at certain places.

A line is considered a control line by the compiler if there is a dollar sign in the left margin, even if it appears to be part of a PL/M-80 comment or character string constant.

On a control line, the dollar sign is followed by zero or more blanks and then by a sequence of controls. The controls must be separated from each other by one or more blanks.

Examples of control lines:

```
$NOCODE XREF
$ EJECT   CODE
```

There are two types of controls: *primary* and *general*. Primary controls must occur either in the invocation command or on a control line which precedes the first noncontrol line of the source file. Primary controls may not be changed within a module. General controls may occur either in the invocation command or on a control line located anywhere in the source input and may be changed freely within a module.

There are a large number of available controls, but few will be needed for most compilations as a set of defaults is built into the compiler. The controls are summarized in a table on the next page.

A *control* consists of a control-name which, depending on the particular control, may be followed by a parenthesized *control parameter*.

Examples of controls:

```
LIST
NOXREF
OBJECT(PROG2.OBJ)
```

3.2 LISTING SELECTION CONTROLS

These controls determine what types of listings are to be produced and on which device they are to appear.

The controls are:

PRINT/NOPRINT
LIST/NOLIST
CODE/NOCODE
XREF/NOXREF
IXREF/NOIXREF
SYMBOLS/NOSYMBOLS

3.2.1 PRINT/NOPRINT

These are primary controls. They have the form:

PRINT[(pathname)]
NOPRINT
Default: PRINT(source-file.LST)

The PRINT control specifies that printed output is to be produced. *Pathname* is a standard ISIS-II pathname which specifies the file or device to receive the printed output. Any output-type device, including a disk file, may be given. If the control is absent, or if a PRINT control appears without a pathname, printed output is directed to the same device used for source input and the output file has the same name as the source file but with the extension LST.

Example:

```
PRINT(:LP:)
```

This causes printed output to be directed to the line printer.

The NOPRINT control specifies that no printed output is to be produced, even if implied by other listing controls such as LIST and CODE.

PRIMARY CONTROL NAMES		DEFAULT
PRINT	NOPRINT	PRINT(source-file.LST)
OBJECT	NOOBJECT	OBJECT(source-file.OBJ)
SYMBOLS	NOSYMBOLS	NOSYMBOLS
XREF	NOXREF	NOXREF
PAGING	NOPAGING	PAGING
DEBUG	NODEBUG	NODEBUG
OPTIMIZE	NOOPTIMIZE	OPTIMIZE
DATE		no date
TITLE		no title
PAGEWIDTH		PAGEWIDTH(120)
PAGELENGTH		PAGELENGTH(60)
INTVECTOR	NOINTVECTOR	INTVECTOR(8,0)
WORKFILES		WORKFILES(:F1:,:F1:)
IXREF	NOIXREF	NOIXREF
GENERAL CONTROL NAMES		**DEFAULT**
LIST	NOLIST	LIST
CODE	NOCODE	NOCODE
EJECT		–
INCLUDE		–
LEFTMARGIN		LEFTMARGIN(1)

Table 3-1. Compiler Controls

3.2.2 LIST/NOLIST

These are general controls. They have the form:

LIST
NOLIST
Default: LIST

The LIST control specifies that listing of the source program is to resume with the next source line read.

The NOLIST control specifies that listing of the source program is to be suppressed until the next occurrence, if any, of a LIST control.

When LIST is in effect, all input lines (from the source file or from an INCLUDE file), including control lines, are listed. When NOLIST is in effect, only source lines associated with error messages are listed.

Note that the LIST control *cannot* override a NOPRINT control. If NOPRINT is in effect, no listing whatsoever is produced.

3.2.3 CODE/NOCODE

These are general controls. They have the form:

CODE
NOCODE
Default: NOCODE

The CODE control specifies that listing of the generated object code, in standard assembly language format is to begin. This listing is interleaved with the program listing on the listing file.

The NOCODE control specifies that listing of the generated object code is to be suppressed until the next occurrence, if any, of a CODE control.

Note that the CODE control cannot override a NOPRINT control.

3.2.4 XREF/NOXREF

These are primary controls. They have the form:

XREF
NOXREF
Default: NOXREF

The XREF control specifies that a cross-reference listing of source program identifiers is to be produced on the listing file.

The NOXREF control suppresses the cross-reference listing.

Note that the XREF control cannot override a NOPRINT control.

3.2.5 IXREF/NOIXREF

These are primary controls. They have the form:

IXREF[(pathname)]
NOIXREF
Default: NOIXREF

The IXREF control causes an ''intermediate intermodule cross-reference file'' to be produced and written out to the file specified by the pathname. If no pathname is supplied, the file will be written on the same device used for source input and will have the same name as the source file but with the extension IXI.

The intermediate file contains all PUBLIC and EXTERNAL identifiers declared in the module being compiled, together with their types, dimensions, and attributes.

After compilation, the IXREF *program* (which is independent of the compiler) can be used to merge two or more of these intermediate files to produce an intermodule cross-reference listing, as explained in Appendix D.

The NOIXREF control suppresses the production of the intermediate file.

3.2.6 SYMBOLS/NOSYMBOLS

These are primary controls. They have the form:

SYMBOLS
NOSYMBOLS
Default: NOSYMBOLS

The SYMBOLS control specifies that a listing of all identifiers in the PL/M-80 source program and their attributes is to be produced on the listing file.

The NOSYMBOLS control suppresses such a listing.

Note that the SYMBOLS control cannot override a NOPRINT control.

3.3 LISTING FORMAT CONTROLS

These controls determine the format of the listing output of the compiler. The controls are:

PAGING/NOPAGING
PAGELENGTH
PAGEWIDTH
DATE
TITLE
EJECT

3.3.1 PAGING/NOPAGING

These are primary controls. They have the form:

PAGING
NOPAGING
Default: PAGING

The PAGING control specifies that the listed output is to be formatted onto pages. Each page carries a heading identifying the compiler and a page number, and possibly a user specified title and/or date.

The NOPAGING control specifies that page ejecting, page heading, and page numbering are not to be performed. Thus, the listing appears on one long ''page'' as would be suitable for a slow serial output device. If NOPAGING is specified, a page eject is not generated if an EJECT control is encountered.

3.3.2 PAGELENGTH

This is a primary control. It has the form:

PAGELENGTH(length)
Default: PAGELENGTH(60)

where *length* is a non-zero, unsigned integer specifying the maximum number of lines to be printed per page of listing output. This number is taken to include the page headings appearing on a page.

The minimum value for *length* is 4.

3.3.3 PAGEWIDTH

This is a primary control. It has the form:

PAGEWIDTH(width)
Default: PAGEWIDTH(120)

where *width* is a non-zero, unsigned integer specifying the maximum line width, in characters, to be used for listing output.

The minimum value for *width* is 60; the maximum value is 132.

3.3.4 DATE

This is a primary control. It has the form:

DATE(date)
Default: no date

where *date* is any sequence of nine or fewer characters not containing parentheses.

The *date* appears in the heading of all pages of listing output exactly as given in the DATE control.

Example:

```
DATE(25 JAN 77)
```

3.3.5 TITLE

This is a primary control. It has the form:

TITLE('title')
Default: no title

where *title* is a sequence of printable ASCII characters which are enclosed in quotes.

The sequence, truncated on the right if necessary to fit, is placed in the title line of each page of listed output.

The maximum length allowed for *title* is 60 characters, but a narrow pagewidth may restrict this number further.

Example:

```
TITLE('TEST PROGRAM 4')
```

3.3.6 EJECT

This is a general control. It has the form:

EJECT

It causes printing of the current page to terminate and a new page to be started. The control line containing the EJECT control is the first line printed (following the page heading) on the new page.

If the NOPRINT, NOLIST or NOPAGING controls are in effect, the EJECT control is ignored.

3.4 SOURCE FORMAT CONTROLS

There is only one control for specifying the format of the source input. It is:

LEFTMARGIN

3.4.1 LEFTMARGIN

This control has the form:

LEFTMARGIN(column)
Default: LEFTMARGIN(1)

where *column* is a non-zero, unsigned integer specifying the left margin of the source input. All characters to the left of this position on subsequent input lines are not processed by the compiler (but do appear on the listing).

The new setting of the left margin takes effect on the next input line. It remains in effect for all input from the source file and any INCLUDE files until it is reset by another LEFTMARGIN control.

Note that a control line is one that contains a dollar sign in the column specified by the most recent LEFTMARGIN control.

3.5 OBJECT FILE CONTROLS

These controls determine what type of object file is to be produced and on which device it is to appear. The controls are:

INTVECTOR/NOINTVECTOR
OBJECT/NOOBJECT
DEBUG/NODEBUG

3.5.1 INTVECTOR/NOINTVECTOR

These are primary controls. They have the form:

INTVECTOR(interval, location)
NOINTVECTOR
Default: INTVECTOR (8,0)

These controls support the Intel 8259 Programmable Interrupt Controller. If your system does not incorporate the 8259, you need not use either of these controls; the compiler automatically generates an interrupt vector starting at absolute location 0, with an 8-byte interval between entries. (See Section 5.3 for further discussion.)

With the 8259, it may be desirable to specify an absolute location other than 0, and an interval of 4 bytes instead of 8 between entries. The INTVECTOR control makes this possible.

The *interval* in the INTVECTOR control must have a value of either 4 or 8, and may be specified in binary, octal, decimal, or hexadecimal notation (use the notational rules given for PL/M numeric constants in the *PL/M Programming Manual*, 98-268, Section 3.1). This value specifies the number of bytes allocated for each entry in the interrupt vector.

The *location* in the INTVECTOR control must have a value from 0 to 65504 (decimal) and must be a multiple of 32 for a 4-byte *interval* or a multiple of 64 for an 8-byte *interval*.

Like the *interval*, the *location* may be specified in binary, octal, decimal, or hexadecimal notation. In hexadecimal notation, it must have a value from 0 to 0FFE0H and must be a multiple of 20H for a 4-byte *interval* or a multiple of 40H for an 8-byte *interval*.

Alternatively, it may be desirable to create the interrupt vector independently, using either PL/M or assembly language. In this case, the NOINTVECTOR control is used and the compiler does not generate any interrupt vector. The implications of this are discussed in Section 5.3.3.

3.5.2 OBJECT/NOOBJECT

The OBJECT and NOOBJECT controls are primary controls. They have the form:

OBJECT[(pathname)]
NOOBJECT
Default: OBJECT(source-file.OBJ)

The OBJECT control specifies that an object module is to be created during the compilation. The *pathname* is a standard ISIS-II pathname which specifies the file to receive the object module. If the control is absent, or if an OBJECT control appears without a pathname, the object module is directed to the same device and file name as used for source input, but with the extension OBJ.

Example:

```
OBJECT(:F1:OTHER.OBJ)
```

This would cause the object code to be written to the file :F1:OTHER.OBJ.

The NOOBJECT control specifies that an object module is not to be produced.

3.5.3 DEBUG/NODEBUG

These are primary controls. They have the form:

DEBUG
NODEBUG
Default: NODEBUG

The DEBUG control specifies that the object module is to contain the name and relative address of each symbol whose address is known at compile-time, and the statement number and relative address of each source program statement. This information may be used later for symbolic debugging of the source program using the ICE-80 In-Circuit Emulator (see *In-Circuit Emulator/80 Operator's Manual* 98-185).

The NODEBUG control specifies that this information is not to be placed in the object module.

3.6 THE WORKFILES CONTROL

The WORKFILES control is a primary control, with the form:

WORKFILES (:device:,:device:)

Each *device* is the name of a direct access device such as a diskette drive.

During compilation, the compiler creates work files which are deleted at the end of compilation (see Section 2.2.3). If the WORKFILES control is not used, these files will be on :F1:. The WORKFILES control allows you to specify any two devices for storage of these files. For example, to specify storage of work files on Drives 1 and 0, use

```
WORKFILES (:F0:,:F1:)
```

Note that *two* device names are required. To specify only one device, specify it twice—for example, to put all work files on Drive 0, use

```
WORKFILES (:F0:,:F0:)
```

As a rule of thumb, the space required for work files on *each* device is roughly equal to the total space required for the PL/M source (including "included" source files—see Section 3.8 below). If only one device is used for work files, it should have twice this amount of space available.

3.7 OPTIMIZATION CONTROLS (OPTIMIZE/NOOPTIMIZE)

These are primary controls. They have the form:

OPTIMIZE
NOOPTIMIZE
Default: OPTIMIZE

The OPTIMIZE control specifies that the compiler is to perform certain types of optimization on the object program (for example, to eliminate, where possible, repetitive evaluation of identical expressions). The NOOPTIMIZE control specifies that these actions are not to be performed.

3.8 THE SOURCE INCLUSION CONTROL (INCLUDE)

The INCLUDE control has the form:

INCLUDE (pathname)

where *pathname* is a standard ISIS-II pathname specifying a disk file.

Example:

```
INCLUDE(:F1:SYSLIB.SRC)
```

An INCLUDE control must be the rightmost control in a control line or in the invocation command.

The INCLUDE control causes subsequent source lines to be input from the specified file. Input will continue from this file until an end-of-file is detected. At that time, input will be resumed from the file which was being processed when the INCLUDE control was encountered.

An included file may itself contain INCLUDE controls. Note that such nesting of included files may not exceed a depth of *five*.

CHAPTER 4
LISTING FORMATS

4.1 PROGRAM LISTING

During the compilation process a listing of the source input is produced. (Refer to Chapters 2 and 3 for details of the file conventions for this listing.) Each page of the listing carries a numbered page-header which identifies the compiler, and optionally gives a title and/or date. The first part of the listing contains a summary of the compilation beginning with the compiler identification and the name of the PL/M-80 source module being compiled. The next line names the file receiving the object code. Finally, the command line used to invoke the compiler is reproduced. The listing of the program itself follows. A sample program listing (of the program from Chapter 1) is shown in Figure 4-1.

The listing contains a copy of the source input plus additional information. To the left of the source image appear two columns of numbers. The first column provides a sequential numbering of PL/M-80 statements. Error messages when present refer to these statement numbers. The second column gives the block nesting depth of the current statement.

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE MYPROGRAM
OBJECT MODULE PLACED IN :F1:MYPROG.OBJ
COMPILER INVOKED BY: PLM80 :F1:MYPROG.SRC  CODE PAGEWIDTH(68) XREF

 1     MYPROGRAM:  DO;
 2   1   WRITE: PROCEDURE(AFT,BUFFER,LENGTH,STATUS) EXTERNAL;
 3   2            DECLARE(AFT,BUFFER,LENGTH,STATUS) ADDRESS;
 4   2          END WRITE;
 5   1   EXIT:  PROCEDURE EXTERNAL;
 6   2          END EXIT;
 7   1   DECLARE STATUS ADDRESS,
           CRLF LITERALLY '0AH,0DH';    /*CARRIAGE RETUR
     -    N, LINE FEED*/
 8   1   CALL WRITE(0,.('HELLO, THIS IS A PL/M-80 PROGRAM', C
     -   RLF),34,.STATUS);
                                              ;STATEMENT   8
     0022  310000          LXI     SP,@STACK$ORIGIN
     0025  010000          LXI     B,0H
     0028  05              PUSH    B       ; 1
     0029  010000          LXI     B,$-29H
     002C  C5              PUSH    B       ; 2
     002D  110000          LXI     D,STATUS
     0030  012200          LXI     B,22H
     0033  0D0000          CALL    WRITE
 9   1   CALL EXIT;
                                              ; STATEMENT   9
     0036  CD0000          CALL    EXIT
10   1   END MYPROGRAM;
                                              ; STATEMENT   10
     0039  FB              EI
     003A  76              HLT
```

Figure 4-1. Program Listing

Lines included with the INCLUDE control are marked with "=" just to the left of the source image. If the included file contains another INCLUDE control, lines included by this "nested" INCLUDE are marked with "=1". For yet another level of nesting, "=2" is used to mark each line, and so forth up to the compiler's limit of five levels of nesting. These markings make it easy to see where included text begins and ends.

Should a source line be too long to fit on the page in one line it will be continued on the following line. Such continuation lines are marked with "–" just to the left of the source image.

The CODE control may be used to obtain the 8080 assembly code produced in the translation of each PL/M-80 statement. This code listing appears interspersed in the source text in six columns of information conforming to standard assembly language format:

1. Location counter (hexadecimal notation)
2. Resultant binary code (hexadecimal notation)
3. Label field
4. Symbolic operation code
5. Symbolic arguments
6. Comment field

Not all six of these columns will appear on any one line of the code listing. Compiler generated labels (e.g. those which mark the beginning and ending of a DO WHILE loop) are preceded by "@". The comments appearing on PUSH and POP instructions indicate the stack depth associated with the stack instruction.

4.2 SYMBOL AND CROSS-REFERENCE LISTING

If specified by the XREF or SYMBOLS control, a summary of all identifier usage appears following the program listing.

Depending on whether the SYMBOLS or XREF control was used to request the identifier usage summary, five or six types of information are provided in the symbol or cross-reference listing. These are as follows:

1. Statement number where identifier was defined.
2. Address associated with identifier.
3. Size of object identified in bytes.
4. The identifier.
5. Attributes of the identifier.
6. Statement numbers where identifier was referenced (XREF control only).

Notice that a single identifier may be declared more than once in a source module (i.e., an identifier defined twice in different blocks). Each such unique object, even though named by the same identifier, appears as a separate entry in the listing.

The address of the object given is the location of that object relative to the start of its associated segment. Which segment is applicable depends upon the attributes of the object (see Chapter 5).

See Figure 4-2 for an example of the cross-reference listing.

4.3 COMPILATION SUMMARY

Following the listing (or appearing alone if NOLIST is in effect) is a compilation summary. Five pieces of information are provided:

- *Code area size* gives the size in bytes of the *code segment* of the output module.
- *Variable area size* gives the size in bytes of the *data segment* of the output module.
- *Maximum stack size* gives the size in bytes of the *stack segment* allocated for the output module.
- *Lines read* gives the number of source lines processed during compilation.
- *Program errors* gives the number of error messages issued during compilation.

See Figure 4-3 for an example of the compilation summary. Refer to Chapter 5 for an explanation of the various memory segments.

```
CROSS-REFERENCE LISTING

  DEFN  ADDR   SIZE  NAME, ATTRIBUTES, AND REFERENCES

     2 0000H     2  AFT             ADDRESS PARAMETER
                                       3

     2 0000H     2  BUFFER          ADDRESS PARAMETER
                                       3

     7               CRLF          LITERALLY
                                       8

     5 0000H         EXIT          PROCEDURE EXTERNAL(1) STACK=0000H
                                       9

     2 0000H     2  LENGTH          ADDRESS PARAMETER
                                       3

       0000H         MEMORY        BYTE ARRAY(0)

     1 0022H    25  MYPROGRAM       PROCEDURE STACK=0006H

     7 0000H     2  STATUS          ADDRESS
                                       6

     2 0000H     2  STATUS          ADDRESS PARAMETER
                                       3

     2 0000H         WRITE         PROCEDURE EXTERNAL(0) STACK=0000H
```

Figure 4-2. Cross-Reference Listing

```
MODULE INFORMATION:

     CODE AREA SIZE     = 003BH    59D
     VARIABLE AREA SIZE = 0002H     2D
     MAXIMUM STACK SIZE = 0006H     6D
     13 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

Figure 4-3. Compilation Summary

CHAPTER 5
RUN-TIME CONVENTIONS

This section presents the run-time organization of PL/M-80 programs in an 8080 CPU environment.

5.1 STORAGE ALLOCATION

Memory for a PL/M-80 program is allocated in four independently relocatable segments.

- Code Segment
- Data Segment
- Stack Segment
- Memory Segment

5.1.1 CODE SEGMENT

This segment contains the object code generated by the source program and space for variables declared with the DATA attribute (except those located elsewhere due to an AT attribute).

5.1.2 DATA SEGMENT

All variables which are not BASED, located with an AT attribute, initialized with the DATA attribute, or local to a REENTRANT procedure are allocated space in this segment.

Space is allocated contiguously for variables in the order of declaration except for formal parameters to nonREENTRANT procedures. These parameters are allocated space contiguously in the order of appearance in the parameter list and before any local variables of the procedure.

5.1.3 STACK SEGMENT

The stack segment is used for the following:

- System status information during interrupt processing.
- The return address on procedure CALLs.
- Parameters to procedures (excepting the last two which are passed via registers).
- Any temporary storage used by the object program, but not explicitly declared in the PL/M source program.
- Automatic storage, i.e., storage for non-based local variables in REENTRANT procedures. This does not include those which have the DATA or AT attributes. Variables with the DATA attribute are allocated in the code segment.

The exact size of the stack is automatically determined by the compiler except for possible multiple incarnations of reentrant procedures. The user can override this computation of stack size and explicitly state the stack requirement during the LOCATE process.

NOTE

When using reentrant procedures the user must be careful to allocate a stack segment large enough to accommodate all possible storage required by multiple incarnations of such procedures.

5.1.4 MEMORY SEGMENT

This is the area of memory which remains after the allocation of the other segments. It may be referenced by the built-in PL/M-80 identifier MEMORY.

5.2 PROCEDURE AND ASSEMBLY LANGUAGE LINKAGE

Formal parameters declared in a procedure definition are treated as locally-defined variables. That is, each parameter is allocated storage sequentially in memory as if it were a variable local to the procedure. During procedure invocation, actual parameters are evaluated, and the results assigned to the corresponding formal parameters.

- A single BYTE parameter is passed in register C. A single ADDRESS parameter is passed in registers B (high-order byte) and C (low-order byte).
- If there are two parameters, the first is passed as described above; the second is passed in registers D (high-order byte, if any) and E (low-order byte).
- When there are more than two parameters, the last two are sent as described above, and the remainder are passed via the stack.

All actual parameters are copied into the locations of the formal parameters on entry to the procedure.

CPU registers are used to hold results returned by procedures which have the BYTE or ADDRESS attribute. In the case of a BYTE procedure, the value returned is in the A register, while an ADDRESS procedure returns the low-order byte in register L and the high-order byte in register H.

Linkage to assembly language subroutines should follow the same conventions as those for PL/M-80 procedures.

5.3 INTERRUPT PROCESSING

5.3.1 SYSTEM WITHOUT 8259

In a system that does not incorporate the 8259 Programmable Interrupt Controller, an interrupt is initiated when the CPU accepts an RST *n* instruction from a peripheral device.

Note that the RST *n* will not be accepted unless interrupts are enabled. The CPU starts with interrupts disabled.

The RST *n* causes the following actions:

1. Interrupts are disabled.
2. The program counter (PC) is saved on the stack.
3. Control passes to absolute location 8**n*.

The location 8**n* is assumed to be the beginning of the interrupt vector entry for a PL/M procedure that was declared with the INTERRUPT *n* attribute. The entry, automatically generated by the compiler, is a JMP instruction (unconditional branch) to the beginning of the code for the procedure.

At the beginning of the interrupt procedure, the compiler automatically inserts code to save CPU register contents on the stack, in the following sequence:

1. (H,L)
2. (D,E)
3. (B,C)
4. (A,Flags)

The procedure body is then executed, as specified in the program.

The automatically generated code for returning from the interrupt procedure does the following:

1. Restores all CPU register contents (except PC) from the stack.
2. Enables interrupts.
3. Restores the program counter from the stack, where it was saved by the RST *n* instruction. This causes control to be transferred back to the point where the interrupt occurred.

All of the above applies when the system does not have the 8259 and no INTVECTOR or NOINTVECTOR controls are used.

5.3.2 THE INTVECTOR CONTROL AND THE 8259

The INTVECTOR control allows you to specify a *location* and an *interval* for the interrupt vector (see Section 3.5.1). For a procedure with the INTERRUPT *n* attribute, the entry in the interrupt vector is at *location* + *n* * *interval*. (If *location* = 0 and *interval* = 8, this formula reduces to 8**n* as described in the preceding section.)

An interrupt vector with a *location* other than 0 or an *interval* other than 8 can only be used in a system incorporating an 8259 which has been programmed with the same *location* and *interval*. Instead of an RST *n* instruction, the 8259 sends the CPU a 3-byte CALL instruction.

The effect is exactly as described in the preceding section, except that control is initially transferred to *location* + *n* * *interval* instead of 8**n*.

Note that in most cases, it is desirable to specify an *interval* of 4 instead of 8, as this reduces the space required by a factor of 2. The JMP instruction contained in the entry requires only 3 bytes.

5.3.3 WRITING INTERRUPT VECTORS SEPARATELY

In some cases, either with or without the 8259, it may be desirable to write the interrupt vector separately (in PL/M or assembly language). This can be done by using NOINTVECTOR to prevent generation of an interrupt vector by the compiler. The separately created interrupt vector can then be linked into the program.

Remember that if the 8259 is not used, the interrupt vector *must* begin at absolute location 0 and the interval between entries *must* be 8 bytes.

If the 8259 is used, the interval between entries *must* be either 4 or 8 bytes. The beginning of the interrupt vector *must* be at a multiple of 32 (20H) for a 4-byte interval or 64

(40H) for an 8-byte interval. Also, the beginning of the interrupt vector must be between 0 and 65504 (0FFE0H), inclusive.

The usefulness of a separately created interrupt vector can be seen by considering an example.

Suppose that two modules for a multimodule program are developed separately. Both use interrupt procedures, but at the time when the modules are written the assignment of interrupt numbers to the various interrupt procedures has not been determined.

The two modules are therefore compiled with the NOINTVECTOR control. When this is done, the *n* in an INTERRUPT *n* attribute is ignored—since normally it would only be used to put the procedure's entry in the proper location within the interrupt vector.

Later, when the program is linked together, a separately created interrupt vector can be linked in. Within this interrupt vector, the placement of the entry for a given interrupt procedure determines which interrupt number will activate that procedure.

Similarly, you could have a library of interrupt procedures, all compiled with NOINTVECTOR. Any program could then have any of these procedures linked in, with a separately created interrupt vector.

5.4 COMPILER-GENERATED OUT-OF-LINE ROUTINES

In generating machine code from a PL/M program, the compiler programs many operations as *out-of-line routines*. An out-of-line routine is much like a procedure: it is written only once, and can be called from other points in the program. For example, some of the operators in PL/M expressions are implemented by out-of-line routines.

This approach significantly reduces the amount of code generated. For example, the actual code for a / (division) operator appears only once, even though the PL/M program may contain many / operators.

The handling of out-of-line routines depends on whether the module being compiled is a single-module program or part of a multimodule program, as explained in the following sections.

5.4.1 SINGLE-MODULE PROGRAMS

The compiler considers a PL/M module to be a single-module program if *both* of the following are true:

- The module is a main program module—i.e., it contains executable statements at the module level.
- It does not contain any PUBLIC or EXTERNAL declarations.

When both of these are true, the compiler places the out-of-line routines in the compiled object file.

In the code listing, the out-of-line routines can be identified by the fact that their names begin with the characters @P.

5.4.2 MULTIMODULE PROGRAMS

If the module being compiled is not a main program module, or if it contains any PUBLIC or EXTERNAL declarations, the compiler assumes that it belongs to a multimodule program.

In this case, the compiler does *not* place the out-of-line routines in the compiled object file. Instead, it treats them like EXTERNAL procedures to be linked in from another file.

Therefore, when a multimodule program is linked together, a file containing the actual code for the out-of-line routines must *always* be linked in, as this code will not be in any of the compiled modules.

The code for all compiler-generated out-of-line routines is available in a library file named PLM80.LIB, which is supplied with the compiler. This file may be linked into a program, or the ISIS-II LIB facility can be used to merge it with another library which will be linked into the program.

NOTE

PLM80.LIB (or a library into which PLM80.LIB has been merged) must be the *last* file in the input list for the LINK facility.

The use of LIB with the PLM80.LIB file is facilitated by the fact that all out-of-line routine names begin with the characters @P.

This handling of out-of-line routines for multimodule programs keeps the program size down by guaranteeing that each out-of-line routine appears only once in the entire program.

5.4.3 A SPECIAL CASE

There is a special (and unusual) case where a program has more than one module, but the main program module does not contain any EXTERNAL or PUBLIC declarations. This could occur if the other modules contained only interrupt procedures.

In this case you should introduce a PUBLIC or EXTERNAL attribute into the main program module.

NOTE

It is enough to make one variable PUBLIC; this will not affect the operation of the program.

Otherwise, the compiler would treat the main program module as a single-module program, placing the out-of-line routines in the object file. These out-of-line routines would not be accessible to other modules in the program, so you would still have to link in a library file containing out-of-line routines for these other modules.

This would result in duplication of some or all of the out-of-line routines, which would be a waste of space.

APPENDIX A

PROGRAM SIZE CONSTRAINTS

Certain fixed size tables within the compiler constrain various features of a user program to certain maximums. These limits are summarized below:

MAXIMUM:

Number of elements in a factored declare list	32
Number of members in a structure	32
Number of labels on a statement	9
Number of procedures in a module	255
Number of DO blocks in a procedure	255
Nesting of blocks	18
Length of an input source line (including CR and LF)	122
Length of a string constant	255
Nesting of INCLUDE controls	5

APPENDIX B

ERROR MESSAGES

The compiler may issue five varieties of error messages:

- Source PL/M-80 errors
- Fatal command tail and control errors
- Fatal input/output errors
- Fatal insufficient memory errors
- Fatal compiler failure errors

The source errors are reported in the program listing; the fatal errors are reported on the console device.

B.1 SOURCE PL/M-80 ERRORS

Nearly all of the source PL/M-80 errors are interspersed in the listing at the point of error and follow the general format:

*** ERROR mmm, STATEMENT nnn, NEAR ''aaa'', message

where

- *mmm* is the error number from the list below
- *nnn* is the source statement number where the error occurs
- *aaa* is the source text near where the error is detected
- *message* is the error explanation from the list below

Source error message list:

1. INVALID PL/M-80 CHARACTER
2. UNPRINTABLE ASCII CHARACTER
3. IDENTIFIER, STRING, OR NUMBER TOO LONG, TRUNCATED
4. ILLEGAL NUMERIC CONSTANT TYPE
5. INVALID CHARACTER IN NUMERIC CONSTANT
6. ILLEGAL MACRO REFERENCE, RECURSIVE EXPANSION
7. LIMIT EXCEEDED: MACROS NESTED TOO DEEPLY
8. INVALID CONTROL FORMAT
9. INVALID CONTROL
10. ILLEGAL USE OF PRIMARY CONTROL AFTER NON-CONTROL LINE
11. MISSING CONTROL PARAMETER
12. INVALID CONTROL PARAMETER
13. LIMIT EXCEEDED: INCLUDE NESTING
14. INVALID CONTROL FORMAT, INCLUDE NOT LAST CONTROL
15. MISSING INCLUDE CONTROL PARAMETER
16. ILLEGAL PRINT CONTROL
17. INVALID PATHNAME
18. INVALID MULTIPLE LABELS AS MODULE NAMES
19. INVALID LABEL IN MODULE WITHOUT MAIN PROGRAM
20. MISMATCHED IDENTIFIER AT END OF BLOCK

21. MISSING PROCEDURE NAME
22. INVALID MULTIPLE LABELS AS PROCEDURE NAMES
23. INVALID LABELLED END IN EXTERNAL PROCEDURE
24. INVALID STATEMENT IN EXTERNAL PROCEDURE
25. UNDECLARED PARAMETER
26. INVALID DECLARATION, STATEMENT OUT OF PLACE
*27. LIMIT EXCEEDED: NUMBER OF DO BLOCKS
28. MISSING 'THEN'
29. ILLEGAL STATEMENT
30. LIMIT EXCEEDED: NUMBER OF LABELS ON STATEMENT
*31. LIMIT EXCEEDED: PROGRAM TOO COMPLEX
32. INVALID SYNTAX, TEXT IGNORED UNTIL ';'
33. DUPLICATE LABEL DECLARATION
34. DUPLICATE PROCEDURE DELCARATION
*35. LIMIT EXCEEDED: NUMBER OF PROCEDURES
36. MISSING PARAMETER
37. MISSING ')' AT END OF PARAMETER LIST
38. DUPLICATE PARAMETER NAME
39. INVALID ATTRIBUTE OR INITIALIZATION, NOT AT MODULE LEVEL
40. DUPLICATE ATTRIBUTE
41. CONFLICTING ATTRIBUTE
42. INVALID INTERRUPT VALUE
43. MISSING INTERRUPT VALUE
44. ILLEGAL ATTRIBUTE, 'INTERRUPT' WITH PARAMETERS
45. ILLEGAL ATTRIBUTE, 'INTERRUPT' WITH TYPED PROCEDURE
46. ILLEGAL USE OF LABEL
47. MISSING ')' AT END OF FACTORED DECLARATION
48. ILLEGAL DECLARATION STATEMENT SYNTAX
49. LIMIT EXCEEDED: NUMBER OF ITEMS IN FACTORED DECLARE
50. INVALID ATTRIBUTES FOR BASE
51. INVALID BASE, SUBSCRIPTING ILLEGAL
52. INVALID BASE, MEMBER OF BASED STRUCTURE OR ARRAY OF STRUCTURES
53. INVALID STRUCTURE MEMBER IN BASE
54. UNDECLARED BASE
55. UNDECLARED STRUCTURE MEMBER IN BASE
56. INVALID MACRO TEXT, NOT A STRING CONSTANT
57. INVALID DIMENSION, ZERO ILLEGAL
58. INVALID STAR DIMENSION IN FACTORED DECLARATION
59. ILLEGAL DIMENSION ATTRIBUTE
60. MISSING ')' AT END OF DIMENSION
61. MISSING TYPE
62. INVALID STAR DIMENSION WITH THIS ATTRIBUTE
63. INVALID DIMENSION WITH 'STRUCTURE'
64. MISSING STRUCTURE MEMBERS
65. MISSING ')' AT END OF STRUCTURE MEMBER LIST
66. INVALID STRUCTURE MEMBER, NOT AN IDENTIFIER
67. DUPLICATE STRUCTURE MEMBER NAME
68. LIMIT EXCEEDED: NUMBER OF STRUCTURE MEMBERS
69. INVALID STAR DIMENSION WITH STRUCTURE MEMBER
70. INVALID MEMBER TYPE, 'STRUCTURE' ILLEGAL
71. INVALID MEMBER TYPE, 'LABEL' ILLEGAL
72. MISSING TYPE FOR STRUCTURE MEMBER
73. INVALID ATTRIBUTE OR INITIALIZATION, NOT AT MODULE LEVEL
74. INVALID STAR DIMENSION, NOT WITH DATA OR INITIAL
75. MISSING ARGUMENT OF 'AT', 'DATA', OR 'INITIAL'

76. CONFLICTING ATTRIBUTE WITH PARAMETER
77. INVALID PARAMETER DECLARATION, BASE ILLEGAL
78. DUPLICATE DECLARATION
79. ILLEGAL PARAMETER TYPE, NOT BYTE OR ADDRESS
80. INVALID DECLARATION, 'LABEL' MAY NOT BE BASED
81. CONFLICTING ATTRIBUTE WITH 'BASE'
82. INVALID SYNTAX, MISMATCHED '('
*83. LIMIT EXCEEDED: DYNAMIC STORAGE
*84. LIMIT EXCEEDED: BLOCK NESTING
85. LONG STRING ASSUMED CLOSED AT NEXT SEMICOLON OR QUOTE
86. LIMIT EXCEEDED: SOURCE LINE LENGTH
87. MISSING 'END', END-OF-FILE ENCOUNTERED
88. INVALID PROCEDURE NESTING, ILLEGAL REENTRANT PROCEDURE
89. MISSING 'DO' FOR MODULE
90. MISSING NAME FOR MODULE
91. ILLEGAL PAGELENGTH CONTROL VALUE
92. ILLEGAL PAGEWIDTH CONTROL VALUE
93. MISSING 'DO' FOR 'END', 'END' IGNORED
94. ILLEGAL CONSTANT, VALUE > 65535
95. ILLEGAL RESPECIFICATION OF PRIMARY CONTROL IGNORED
96. COMPILER ERROR: SCOPE STACK UNDERFLOW
97. COMPILER ERROR: PARSE STACK UNDERFLOW
*98. INCLUDE FILE IS NOT A DISKETTE FILE
100. INVALID STRING CONSTANT IN EXPRESSION
101. INVALID ITEM FOLLOWS DOT OPERATOR
102. MISSING PRIMARY OPERAND
103. MISSING ')' AT END OF SUBEXPRESSION
104. ILLEGAL PROCEDURE INVOCATION WITH DOT OPERATOR
105. UNDECLARED IDENTIFIER
106. INVALID INPUT
107. ILLEGAL INPUT
108. MISSING ')' AFTER INPUT
109. MISSING INPUT
110. INVALID LEFT OPERAND OF QUALIFICATION, NOT A STRUCTURE
111. INVALID RIGHT OPERAND OF QUALIFICATION, NOT IDENTIFIER
112. UNDECLARED STRUCTURE MEMBER
113. MISSING ')' AT END OF ARGUMENT LIST
114. INVALID SUBSCRIPT, MULTIPLE SUBSCRIPTS ILLEGAL
115. MISSING ')' AT END OF SUBSCRIPT
116. MISSING '=' IN ASSIGNMENT STATEMENT
117. MISSING PROCEDURE NAME IN CALL STATEMENT
118. INVALID INDIRECT CALL, IDENTIFIER NOT AN ADDRESS SCALAR
*119. LIMIT EXCEEDED: PROGRAM TOO COMPLEX
*120. LIMIT EXCEEDED: EXPRESSION TOO COMPLEX
*121. LIMIT EXCEEDED: EXPRESSION TOO COMPLEX
*122. LIMIT EXCEEDED: PROGRAM TOO COMPLEX
123. INVALID DOT OPERAND, BUILT-IN PROCEDURE ILLEGAL
124. MISSING ARGUMENTS FOR BUILT-IN PROCEDURE
125. ILLEGAL ARGUMENT FOR BUILT-IN PROCEDURE
126. MISSING ')' AFTER BUILT-IN PROCEDURE ARGUMENT LIST
127. INVALID SUBSCRIPT ON NON-ARRAY
128. INVALID LEFT-HAND OPERAND OF ASSIGNMENT
129. ILLEGAL 'CALL' WITH TYPED PROCEDURE
130. ILLEGAL REFERENCE TO OUTPUT FUNCTION
131. ILLEGAL REFERENCE TO UNTYPED PROCEDURE

132. ILLEGAL USE OF LABEL
133. ILLEGAL REFERENCE TO UNSUBSCRIPTED ARRAY
134. ILLEGAL REFERENCE TO UNSUBSCRIPTED MEMBER ARRAY
135. ILLEGAL REFERENCE TO AN UNQUALIFIED STRUCTURE
136. INVALID RETURN FOR UNTYPED PRODECURE, VALUE ILLEGAL
137. MISSING VALUE IN RETURN FOR TYPED PROCEDURE
138. MISSING INDEX VARIABLE
139. INVALID INDEX VARIABLE TYPE, NOT BYTE OR ADDRESS
140. MISSING '=' FOLLOWING INDEX VARIABLE
141. MISSING 'TO' CLAUSE
142. MISSING IDENTIFIER FOLLOWING GOTO
143. INVALID REFERENCE FOLLOWING GOTO, NOT A LABEL
144. INVALID GOTO LABEL, NOT AT LOCAL OR MODULE LEVEL
145. MISSING 'TO' FOLLOWING 'GO'
146. MISSING ')' AFTER 'AT' RESTRICTED EXPRESSION
147. MISSING IDENTIFIER FOLLOWING DOT OPERATOR
148. INVALID QUALIFICATION IN RESTRICTED REFERENCE
149. INVALID SUBSCRIPTING IN RESTRICTED REFERENCE
150. MISSING ')' AT END OF RESTRICTED SUBSCRIPT
151. INVALID OPERAND IN RESTRICTED EXPRESSION
152. MISSING ')' AFTER CONSTANT LIST
153. INVALID NUMBER OF ARGUMENTS IN CALL, TOO MANY
154. INVALID NUMBER OF ARGUMENTS IN CALL, TOO FEW
155. INVALID RETURN IN MAIN PROGRAM
156. MISSING RETURN STATEMENT IN TYPED PROCEDURE
157. INVALID ARGUMENT, ARRAY REQUIRED FOR LENGTH OR LAST
158. INVALID DOT OPERAND, LABEL ILLEGAL
159. COMPILER ERROR: PARSE STACK UNDERFLOW
160. COMPILER ERROR: OPERAND STACK UNDERFLOW
161. COMPILER ERROR: ILLEGAL OPERAND STACK EXCHANGE
162. COMPILER ERROR: OPERATOR STACK UNDERFLOW
163. COMPILER ERROR: GENERATION FAILURE
164. COMPILER ERROR: SCOPE STACK OVERFLOW
165. COMPILER ERROR: SCOPE STACK UNDERFLOW
166. COMPILER ERROR: CONTROL STACK OVERFLOW
167. COMPILER ERROR: CONTROL STACK UNDERFLOW
168. COMPILER ERROR: BRANCH MISSING IN 'IF' STATEMENT
169. ILLEGAL FORWARD CALL
170. ILLEGAL RECURSIVE CALL
171. INVALID USE OF DELIMITER OR RESERVED WORD IN EXPRESSION
172. INVALID LABEL: UNDEFINED
173. INVALID LEFT SIDE OF ASSIGNMENT: VARIABLE DECLARED WITH DATA ATTTRIBUTE
174. INVALID NULL PROCEDURE
176. INVALID INTVECTOR INTERVAL VALUE
177. INVALID INTVECTOR LOCATION VALUE
178. INVALID 'AT' RESTRICTED REFERENCE, EXTERNAL ATTRIBUTE CONFLICTS WITH PUBLIC ATTRIBUTE
200. LIMIT EXCEEDED: STATEMENT SIZE
201. INVALID DO CASE BLOCK, AT LEAST ONE CASE REQUIRED
202. LIMIT EXCEEDED: NUMBER OF ACTIVE CASES
203. LIMIT EXCEEDED: NESTING OF TYPED PROCEDURE CALLS
204. LIMIT EXCEEDED: NUMBER OF ACTIVE BLOCKS OR CASES
205. ILLEGAL NESTING OF BLOCKS, ENDS NOT BALANCED
206. LIMIT EXCEEDED: CODE SEGMENT SIZE

207. LIMIT EXCEEDED: SEGMENT SIZE
208. LIMIT EXCEEDED: STRUCTURE SIZE
209. ILLEGAL INITIALIZATION OF MORE SPACE THAN DECLARED
210. ILLEGAL INITIALIZATION OF A BYTE TO A VALUE > 255
211. INVALID VARIABLE IN 'AT' RESTRICTED REFERENCE
212. INVALID RESTRICTED REFERENCE IN 'AT', BASE ILLEGAL
213. UNDEFINED RESTRICTED REFERENCE IN 'AT'
214. COMPILER ERROR: OPERAND CANNOT BE TRANSFORMED
215. COMPILER ERROR: EOF READ IN FINAL ASSEMBLY
216. COMPILER ERROR: BAD LABEL ADDRESS
217. ILLEGAL INITIALIZATION OF AN EXTERNAL VARIABLE
218. ILLEGAL SUCCESSIVE USES OF RELATIONAL OPERATORS

NOTE

The error messages flagged with * in the list above are terminal errors. If such an error is encountered program text beyond the point of error is not compiled. A terminal error message will appear at the beginning of the program listing and at the point of error in the program listing.

B.2 FATAL COMMAND TAIL AND CONTROL ERRORS

Fatal command tail errors are caused by an improperly specified compiler invocation command or an improper control. The errors which may occur here are as follows:

ILLEGAL COMMAND TAIL SYNTAX OR VALUE
UNRECOGNIZED CONTROL IN COMMAND TAIL
INCLUDE FILE IS NOT A DISKETTE FILE
INVOCATION COMMAND DOES NOT END WITH <CR><LF>
INCORRECT DEVICE SPECIFICATION
SOURCE FILE NOT A DISKETTE FILE
SOURCE FILE NAME INCORRECT
SOURCE FILE EXTENSION INCORRECT
ILLEGAL COMMAND TAIL SYNTAX
MISPLACED CONTROL: WORKFILES ALREADY OPENED

B.3 FATAL INPUT/OUTPUT ERRORS

Fatal input/output errors occur when the user incorrectly specifies a pathname for compiler input or output. These error messages are of the form:

PL/M-80 I/O ERROR —
FILE:
NAME:
ERROR:
COMPILATION TERMINATED

The errors that may occur here are as follows:

ILLEGAL FILENAME SPECIFICATION
ILLEGAL OR UNRECOGNIZED DEVICE SPECIFICATION IN FILENAME
ATTEMPT TO OPEN AN ALREADY OPEN FILE

NO SUCH FILE
FILE IS WRITE PROTECTED
FILE IS NOT ON A DISKETTE
DEVICE TYPE NOT COMPATIBLE WITH INTENDED USE
FILENAME REQUIRED ON DISKETTE FILE
NULL FILE EXTENSION
ATTEMPT TO READ PAST EOF

B.4 FATAL INSUFFICIENT MEMORY ERRORS

The fatal insufficient memory errors are caused by a system configuration with not enough RAM memory to support the compiler.

The errors that may occur due to insufficient memory are as follows:

NOT ENOUGH MEMORY FOR COMPILATION
DYNAMIC STORAGE OVERFLOW
NOT ENOUGH MEMORY

B.5 FATAL COMPILER FAILURE ERRORS

The fatal compiler failure errors are internal errors that should never occur. If you encounter such an error, please report it to Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051, Attn: Software Marketing Department.

The errors falling into this class are as follows:

SYNC FAILURE READING GLOBALS
UNKNOWN FATAL ERROR
96. COMPILER ERROR: SCOPE STACK UNDERFLOW
97. COMPILER ERROR: PARSE STACK UNDERFLOW
159. COMPILER ERROR: PARSE STACK UNDERFLOW
160. COMPILER ERROR: OPERAND STACK UNDERFLOW
161. COMPILER ERROR: ILLEGAL OPERAND STACK EXCHANGE
162. COMPILER ERROR: OPERATOR STACK UNDERFLOW
163. COMPILER ERROR: GENERATION FAILURE
164. COMPILER ERROR: SCOPE STACK OVERFLOW
165. COMPILER ERROR: SCOPE STACK UNDERFLOW
166. COMPILER ERROR: CONTROL STACK OVERFLOW
167. COMPILER ERROR: CONTROL STACK UNDERFLOW
168. COMPILER ERROR: BRANCH MISSING IN 'IF' STATEMENT
214. COMPILER ERROR: OPERAND CANNOT BE TRANSFORMED
215. COMPILER ERROR: EOF READ IN FINAL ASSEMBLY
216. COMPILER ERROR: BAD LABEL ADDRESS

APPENDIX C

PL/M-80 LIBRARY SUPPORT FOR THE 8085 PROCESSOR

The 8085 provides more complex interrupt handling than the 8080. To support the 8085, the library file PLM80.LIB (supplied with the compiler) contains two procedures which correspond to the 8085 instructions Read Interrupt Mask (RIM) and Set Interrupt Mask (SIM).

The library procedure R$MASK is a typed procedure corresponding to the RIM instruction. To link it into a PL/M-80 procedure, use the following declaration in the source code for the main program module:

```
R$MASK:  PROCEDURE BYTE EXTERNAL;
         END R$MASK;
```

A reference to R$MASK — for example

```
INT$CONDITION = R$MASK;
```

returns a byte value whose bits show the current state of the 8085's interrupt mechanism (see description of RIM instruction in the *8085 Microcomputer Systems User's Manual,* 98-366).

The reverse can be achieved with the S$MASK Procedure, which accepts a BYTE parameter and uses its 8 bits to program the 8085's interrupt mechanism. To link S$MASK into a PL/M-80 program, use the following declaration in the source code for the main program module:

```
S$MASK:  PROCEDURE (MASK) EXTERNAL;
         DECLARE MASK BYTE;
         END S$MASK;
```

The procedure can be called to program the 8085's interrupt mechanism as in the following example:

```
CALL S$MASK (INT$CONDITION);
```

where INT$CONDITION is a BYTE variable. The usage of the bits is described under the SIM instruction in the *8085 Microcomputer Systems User's Guide,* 98-366.

APPENDIX D

THE IXREF PROGRAM

D.1 GENERAL

The IXREF program is supplied on the same diskette as the ISIS-II PL/M-80 Compiler, Version 3.1. It uses intermediate files produced by the compiler under the IXREF control (see Section 3.2.5) to produce an intermodule cross-reference file.

To use this facility, first compile all modules that are to be cross-referenced, using the IXREF *control* in each case. Then run the IXREF *program* as explained below.

D.2 INVOKING THE IXREF PROGRAM

The IXREF program invocation command has the following general form:

[:device:] IXREF input-list [controls]

where

- *device* identifies which drive contains the diskette with the IXREF program. This may be omitted if the diskette is in Drive 0.
- *input-list* is a list of pathnames of intermediate files produced by the compiler under the IXREF control. The pathnames must be separated by commas (spaces may also be inserted between pathnames). The pathnames may be in any order and may use the ''wild card'' construction (see ISIS-II System User's Guide, Intel document number 98-306). If any of the specified files is not a valid intermediate file, IXREF will type the pathname and the message BAD RECORD TYPE and will skip the file.
- *controls* is an optional sequence of one or more controls separated by spaces. Controls are described below.

If the invocation command is too long to be typed on one line, you can break it by typing an & character followed by a carriage return. The & must not be within a pathname or control. IXREF responds to the & with a * * prompt to show that it is waiting for a continuation line.

D.3 CONTROLS

The control sequence in the IXREF program invocation is optional. If no controls are used, the output file will have the following characteristics:

- The output pathname will be the same as the first pathname in the input-list, but with the extension IXD.
- No title will be placed at the top of each page.
- All identifiers declared PUBLIC or EXTERNAL will be listed.

Four controls are provided to modify the characteristics of the output file.

D.3.1 THE PRINT CONTROL

This control has the form

PRINT (pathname)

Where *pathname* is a standard ISIS-II pathname to specify the name of the output file.

D.3.2 THE TITLE CONTROL

This control has the form

TITLE ('string')

Where *string* is a sequence of up to 60 characters to be placed at the top of each page of output. If the 60 character limit is exceeded, the string will be truncated on the right.

D.3.3 THE PUBLICS CONTROL

This control has the form

PUBLICS

and specifies that only PUBLIC identifiers are to be represented in the output file.

D.3.4 THE EXTERNALS CONTROL

This control has the form:

EXTERNALS

and specifies that only EXTERNAL identifiers are to be represented in the output file.

D.4 THE IXREF OUTPUT FILE

Figure D-1 shows a typical intermodule cross-reference file produced by IXREF. Note that a ''wild card'' construction was used in the input-list to input all files on Drive 1 with the extension IXI. Controls were used to specify a title and a pathname for the output file.

The file contains two listings, the ''intermodule crossreference listing'' and the ''module directory.'' Both are sorted alphabetically. Note that in the illustration, portions of the intermodule cross-reference listing have been omitted.

Each entry in the intermodule cross-reference listing begins with an identifier in the left column. In the right column, we have the attributes of the identifier, then a semicolon followed by the names of all modules in which it is declared PUBLIC or EXTERNAL.

The first entry after the semicolon is the name of the module in which the identifier is declared PUBLIC. If no PUBLIC declaration is found, the notation ** UNRESOLVED ** appears.

Thus we can see that ACTUALBASEPTR is an ADDRESS variable declared PUBLIC in module MACRO and EXTERNAL in modules SYMSCN and STACK.

```
ISIS-II IXREF, V1.1
INVOKED BY:
-IXREF :F1:*.IXI TITLE('INTER-MODULE CROSS-REFERENCE') &
            PRINT(:F1:ASSEMB.OUT)

INTER-MODULE CROSS-REFERENCE LISTING

NAME.....................ATTRIBUTES;  MODULE NAMES

ACTUALBASEPTR.............ADDRESS;  MACRO SYMSCN STACK
ACTUALBLOCKENDMARKER......BYTE(2);  MACRO
ACTUALDELIMITER...........PROCEDURE BYTE;  MACRO SCNFSM
ACTUALPROTECTED...........BYTE;  MACRO SCNFSM
   .                          .
   .                          .
   .                          .
BLKSTK....................BYTE(17);  PUBLICDCL DRIVE STACK MACRO
BLOCKNO...................BYTE;  ** UNRESOLVED ** ENDLIN
   .                          .
   .                          .
   .                          .
XREFSYMBUF................BYTE(6);  PUBLICDCL ENDLIN DRIVE RELOBJ
                             SCNFSM
XREFSYMBUFPREVIOUS........BYTE(6);  PUBLICDCL SCNFSM
XREFUTILITYNAME...........BYTE(11);  PUBLICDCL DRIVE INIT
ZERO......................ADDRESS;  MACRO STACK SCNFSM
ZEROADDRESS...............ADDRESS;  ASSEMB RELOBJ

MODULE DIRECTORY

MODULE NAME:.............FILE NAME    DISKETTE NAME

ASSEMB...................ASSEMB.SRC   SOURCE
DRIVE....................DRIVE.SRC    DRIVE
ENDLIN...................ENDLIN.SRC   END
INIT.....................INIT.SRC     INIT.OVL
MACRO....................MACRO.SRC    MACRO.SRC
PUBLICDCL................PUBLIC.SRC   DRIVE
RELOBJ...................RELOBJ.SRC   RELOBJ
SCNFSM...................SCNFSM.SRC   SOURCE
STACK....................STACK.SRC    SYMSCN.STK
SYMSCN...................SYMSCN.SRC   SYMSCN.STK
```

Figure D-1. Intermodule Cross-Reference Listing

In the next entry, we see that ACTUALBLOCKENDMARKER is an array of two BYTE elements, declared PUBLIC in module MACRO.

In the module directory, each entry begins with a module name. In the second column, we find the name of the PL/M-80 source file from which the module was compiled, and in the third column we find the name of the diskette where the source file resides. (A diskette is named when it is formatted with the ISIS-II FORMAT command.)

D.5 ERROR CONDITIONS

IXREF detects the following error conditions in the invocation command:

- Incorrect file specifications in input-list or PRINT control (IXREF terminates and produces no output).
- Nonexistent file in input-list (if possible, IXREF skips to next pathname and continues; otherwise it terminates and produces no output).
- Missing parenthesis in PRINT or TITLE control (IXREF terminates and produces no output).
- Misspelled or unknown controls (IXREF terminates and produces no output).
- PUBLICS and EXTERNALS controls used in same invocation of IXREF (IXREF terminates and produces no output).
- Repetition of a control (IXREF terminates and produces no output).

D.6 TEMPORARY FILES USED BY IXREF

While running, IXREF uses the following temporary files:

```
:device:IXIN.TMP
:device:IXOUT.TMP
:device:MODNM.TMP
```

where *device* is the same device specified for the first file in the input-list. These files are deleted when IXREF terminates. Therefore, if you have any files with these names on the same device as the first file in the input-list, you must rename them before running IXREF.

APPENDIX E

DIFFERENCES BETWEEN VERSION 3.1 AND EARLIER VERSIONS

Version 3.1 of the ISIS-II PL/M-80 Compiler supports intermodule cross-reference listings for multimodule programs. Two new features make this possible:

- A new compiler control, IXREF, causes the compiler to produce an intermediate file when a module is compiled.
- A separate program, also called IXREF, is supplied with the compiler. After the modules of a program are compiled with the IXREF control, the IXREF program combines the intermediate files to produce an intermodule cross-reference listing.

In addition, the compiler software has been improved in various ways which do not affect user control.

Also, note that Section 3.6 describes the WORKFILES control. Sections 3.5.1, 5.3.2, and 5.3.3 describe the INTVECTOR and NOINTVECTOR controls. Section 5.4 describes compiler-generated out-of-line routines and use of the PLM80.LIB library file. Appendix C describes library support for the 8085 processor. These features were introduced in Version 3.0.

INDEX